LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-648664

Characterizing the Impact of Program Optimizations on Power and Energy for Explicit Hydrodynamics

E. A. Leon, I. Karlin

January 22, 2014

International Parallel & Distributed Processing Symposium
Phoenix, AZ, United States
May 19, 2014 through May 23, 2014

Disclaimer

Characterizing the Impact of Program Optimizations on Power and Energy for Explicit Hydrodynamics

Edgar A. León and Ian Karlin
Lawrence Livermore National Laboratory
{*leon,karlin1*}*@llnl.gov*

***Abstract*—With the end of Denard scaling, future systems will be constrained by power and energy. This will impact application developers by forcing them to restructure and optimize their algorithms in terms of these resources. In this paper, we analyze the impact of different code optimizations on power, energy, and execution time. Our optimizations include loop fusion, data structure transformations, global allocation, and compiler selection. We analyze the static and dynamic components of power and energy as applied to the processor chip and memory domains within a system. In addition, our analysis correlates energy and power changes with performance events and shows that data motion is highly correlated with memory power and energy usage and executed instructions are partially correlated with processor power and energy. Our results demonstrate key tradeoffs among power, energy, and execution time for explicit hydrodynamics via a representative kernel. In particular, we observe that loop fusion and compiler selection improve all objectives, while global allocation and data layout transformations present tradeoffs that are objective-dependent.**

***Keywords*-power and energy tuning; static and dynamic power; performance analysis.**

I. Introduction

With the power draw of supercomputers increasing with every generation, power usage is becoming a larger fraction of the total cost of ownership for major computing centers. Because of the cost of facility upgrades to increase power into a computer and as dark silicon becomes a feature of more computing chips, the amount of power that a machine can consume will often exceed the amount of power that can safely be delivered to it. In light of these changes, it is likely that computer centers will begin to charge for both node hours and kilowatt hours. With power monitoring features being added to modern processors such as Intel's RAPL [1] and IBM's Environment Monitoring [2], application programmers are begining to get the fine grain measurement tools they need to measure and tune for power and energy usage.

In recent literature, researchers are investigating the tradeoffs between power, energy, and performance and how these interact on modern systems [2]–[5]. Other studies are exploring how auto-tuning techniques can reduce energy or power consumption [6]. However, most of the literature focuses on total power and energy usage ignoring that since the end of Denard scaling the static or leakage power consumed by computers has increased as a percentage of overall system power [7]. As we will show, the impact of program transformations is not always evident when analyzing overall or total power.

In this paper we focus on analyzing frequently-used program transformations and their effect on power and energy. We further break down power and energy usage into the components of a node that consume it: the memory and compute units. We start by analyzing total power and energy but since static power is constant and static energy is a function of runtime, we focus on the dynamic parts. Our characterization of dynamic power and energy at a component level (e.g., memory) allows us to understand how various program transformations impact power usage within the system and correlate them to performance through changes in hardware counter values. Using a full factorial experimental design of performance tuning of a proxy application representative of explicit hydrodynamics codes at Lawrence Livermore National Laboratory (LULESH), we make the following contributions:

- We show the effects of code transformations on the dynamic power and energy used by various system components.
- We corrolate energy and power changes with performance counters and show what is important when trying to reduce power and energy usage.
- We explore the tradeoffs of tuning LULESH for dynamic energy, runtime, and dynamic power. We also show that changing compilers and loop fusion help with all objectives and when to use data allocation and data transformations is objective dependent.

The rest of this paper is organized as follows. Section II gives an overview of the LULESH proxy application and the optimizations we analyze. Section III describes our experimental setup and methodology. Section IV shows how power and energy are used throughout the system as we optimize LULESH. We break down values into system components and static and dynamic quantities. Section V analyzes how performance optimizations impact dynamic power and energy usage and the tradeoffs between runtime, power, and energy. Section VI analyzes our results and discusses considerations about the profitability of tuning for

power and energy on future systems. Section VII presents an overview of the related work. Finally, Section VIII contains conclusions and areas of future work.

II. LULESH

The Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) mini-app was originally developed as one of the five challenge problems for the DARPA UHPC program. It was choosen because explicit hydrodynamics can consume up to one third of the compute cycles at DOD data centers. The LULESH mini-app provides a simplified source code that contains the data access patterns and computational characteristics of larger hydrodynamics codes. To that end it uses an unstuctured hexadral mesh with two centerings and solves the Sedov problem.

Because of its smaller size, it allows for easier and faster performance tuning experiments on various architectures. The successful lessons learned can then be applied back to larger production codes [8]. Also, LULESH has been ported to a wide variety of programming models to explore their various performance and productivity advantages [9]. LULESH was recently updated to be more representative of production codes at LLNL with the addition of multi-region physics, artificial load imbalance, and various computer science changes. LULESH is currently being used in the Department of Energy's Extreme-Scale Technology Acceleration program (FastForward[1]), the CORAL procurement [2], and the ExMatEx materials co-design center[3].

A. Algorithmic Optimizations

Since this paper focuses on how optimizations impact performance, power, and energy, we employ performance optimizations resulting from previous work [8]. These optimizations include loop fusion, data layout transformations, global allocation, and vectorization. Combined, these optimizations may improve performance up to 3X on conventional microprocessors and up to 10X on accelerators such as the Intel Xeon Phi. In this paper we focus on loop fusion, data layout transformations, global allocation optimizations, and code generation from different compilers.

Loop fusion is an optimization that combines multiple loops with the same iteration space together. When loops that access the same arrays are combined, the amount of data needed to move through the memory hierarchy is reduced [10]. However, fusing loops can decrease performance due to increased register pressure, conflict or capacity misses in cache or other resource constraints [11]. The version of LULESH we use for our fused code contains 12 loops from the original 45. Fusing loops further would result in transformations that are not possible in the codes modeled by the proxy application [8].

[1]https://asc.llnl.gov/fastforward/

[2]https://asc.llnl.gov/CORAL-benchmarks/

[3]http://codesign.lanl.gov/projects/exmatex/index.html

Data layout transformations involve changing a 'struct' of arrays to an array of 'structs'. These transformations can reduce the amount data moved through the memory hierarchy by combining acesses to indirectly accessed data structures [12]. Also, they can reduce the number of streams being prefetched from memory resulting in more effective use of hardware stream prefetchers. In LULESH we combine arrays into 10 different structures. Table I shows the arrays we combine for these optimizations.

Table I
TRANSFORMED DATA STRUCTURES.

Description	Arrays
Coordinates	x, y, z
Velocities	xd, yd, zd
Accelerations	xdd, ydd, zdd
Forces	fx, fy, fz
Principle Stains	dxx, dyy, dzz
Velocity Gradient	delv_xi, delv_eta, delv_zeta
Coordinate Gradient	delx_xi, delx_eta, delx_zeta
Temporary Forces	fx_elem, fy_elem, fz_elem
Presure and Q	p, q
Q Terms	ql, qq

Global allocation for LULESH involves moving all the 'malloc' and 'free' statements outside of the timestep loop. Therefore, all temporary variables are allocated once and then reused without freeing space throughout the program. Another option that can result in a similar performance gain is using a thread aware allocation library such as *tcmalloc*. In some cases these libraries can result in the same performance as global allocation without the programming challenges or memory costs needed to maintain global temporary variables.

III. EXPERIMENTAL SETUP

Our experiments consist of executing multiple versions of LULESH, each representing a different set of optimizations, on an IBM Blue Gene/Q (BG/Q) system capable of power and energy measurements. For each experiment, we capture energy, power, execution time, and several performance counters. In this section, we describe the machine architecture used for our experiments, including the infrastructure to measure power and energy, and the execution environment for LULESH.

A. Machine Architecture

A BG/Q system is organized into racks, midplanes, node boards, and compute nodes [13]. A rack consists of 2 midplanes and each midplane consists of 16 node boards. The connection between midplanes is achieved via link chips (BQL) that reside on the node boards. A node board, or simply board, has 32 compute nodes. Each node has an ASIC with 18 PowerPC A2 cores (BQC), SDRAM-DDR3 memory, and runs IBM's Compute Node Kernel (CNK). Sixteen cores are dedicated for application processing, one for

the operating system, and one unused for redundancy. Each core is a 4-way SMT unit and runs at 1.6 GHz. Network communication on this system is achieved through high-speed serial (HSS) cores on the BQC and BQL components.

B. Power Infrastructure

BG/Q systems are capable of measuring power and energy at a node board granularity [2]. Each board has two direct current amperage (DCA) modules that produce high-voltage power lines, which are translated into low power lines or *domains* by voltage transformation modules. This organization results in a total of seven power domains as shown in Table II. Domain one, for example, includes the L2 cache, the processor units, integrated memory controllers, and the chip-to-chip communication logic. Throughout the rest of the paper we use the named alias (e.g., *core* and *memory*) to refer to the power domains.

Table II
BG/Q POWER DOMAINS PER NODE BOARD.

	Description	Alias
Domain 1	BQC core logic power	Core
Domain 2	SDRAM-DDR3, BQC DDR3 I/O	Memory
Domain 3	Optical module power	Network
Domain 4	Optical module power, PCI Express	
Domain 6	BQC and BQL HSS I/O	
Domain 8	BQL core power	
Domain 7	BQC core array power	

Each DCA has a microprocessor unit (MPU) that measures current and voltage for each domain. Then, an FPGA on the node board reads the measured data from the MPU to its on-chip memory. Finally, the compute nodes can read data from the FPGA via Environment Monitoring (EnvMon) serial bus lines. User-level programs can access the power information through the EMON API [14], [15].

The default BG/Q power infrastructure provides access to instantaneous power measurements at 300 ms intervals. This interval may be too coarse to capture power phases of many applications and, thus, IBM refined its power monitoring on BG/Q to allow measurements of current and voltage at fine-grain intervals that range between 90 and 500 us depending on the number of domains being measured. The new infrastructure requires changes to the DCAs and FPGA images of a node board and provides a new API called EMON2 [2]. All of our investigations in this paper make use of this new infrastructure.

Through the EMON2 API a user application is able to retrieve cumulative energy consumption at a given point in time. With two snapshots, the EMON2 library computes the energy difference and the average power consumption for the given interval. Because of the long latency to retrieve a power snapshot from the FPGA [2], our experiments capture power and energy consumption every 10 ms. We collect data for all seven power domains.

To capture power and energy measurements in tandem with performance counters, we employ the IBM BGQT library [2]. This library leverages the EMON2 and the Blue Gene performance monitoring (BGPM) application programming interfaces. BGQT links to an application during compilation and, at runtime, launches a tracer thread per board to capture power snapshots every time interval (10 ms). Since the latency of collecting a power snapshot can be large, and thus affect application performance, we dedicate one core for the tracing thread.

C. Experimental Methodology

To study individual optimizations and their combined effect on power, energy, and performance, we developed 16 versions of LULESH representing all optimization combinations (see Table III). We compare the different optimizations relative to the unoptimized version of LULESH 1.0 built with the GNU compiler (NoOpt).

Table III
LULESH OPTIMIZATIONS (2^4 VERSIONS).

ac	Global allocation
dl	Data structure layout transformations
fu	Loop fusion
xl	xlc++ compiler with -O3 -qhot -qstrict -qsmp=omp
NoOpt	g++ compiler with -O3 -fopenmp

We ran each version of LULESH on 1 node board (32 nodes) and 60 OpenMP threads per node. We used a problem size of 120^3 per node for 130 iterations and started measurements after the application's initialization phase. To understand the static and dynamic power consumption of LULESH, we executed a single thread benchmark that performs no work. The power consumed by this benchmark (per unit of time) represents the static power. To calculate the dynamic power of a LULESH instance we subtracting this static power measurement from the total power consumed by LULESH.

IV. THE EFFECTS OF OPTIMIZATION ON SYSTEM POWER AND ENERGY USAGE

To understand the effects of optimization on power and energy usage one must understand how power and energy are used throughout the system. In Figure 1 we present a breakdown of the power (an energy breakdown would look identical) used by the unoptimized version of LULESH. Power is broken down by domain including *core*, *memory*, and *other*. The domain named other includes the network domain and the BQC core array (see Table II). The network and BQC core array do not vary significantly as we optimize LULESH so we ignore them for the remainder of the paper. Network links, for example, are always *on*

consuming roughly the same power regardless of use. Then within each domain we show the dynamic and static power consumed. From Figure 1 we observe that static power is the dominant consumer of system power on BG/Q. In addition, the memory subsystem has a larger relative power change when active, although it consumes a smaller portion of the total overall power.

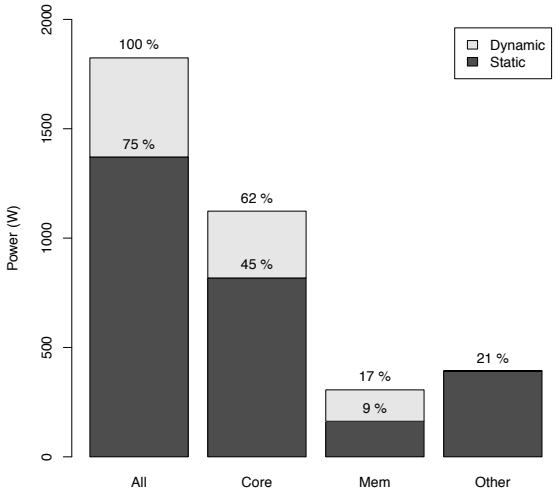

Figure 1. Node board power consumed by the NoOpt configuration itemized by component. This graph shows, for example, that 45% of the total power is static core power, while 17% of the total power is memory power.

To tune codes for power, optimizations must save a significant amount of it to tradeoff these savings with runtime. Figure 2 shows how energy and power change with respect to the optimizations used and runtime. This figure demonstrates that energy use is highly correlated with runtime. Therefore, tuning for runtime and tuning for energy present nearly identical tradeoffs. Also, total power usage never changes by more than 5% so tuning for power has a marginal impact.

However, when we consider the dynamic components of energy and power, optimizations have a greater impact. Figure 3 shows that although a correlation between energy and runtime still exists, the effects are less pronounced. Figure 3 also shows that power variations from performance tuning are significantly greater when static power is removed. Therefore, by removing static power and/or energy from consideration we observe the impact of tuning on what a programmer can control.

The final components we consider to understand tuning tradeoffs on a system are the changes in power and energy at a subsystem level, i.e., the memory and core domains (see Figures 4 and 5). In more than half the code versions in Figure 4, power from one component (most frequently memory power) decreases relative to the untuned code while power from the other component increases. In these cases,

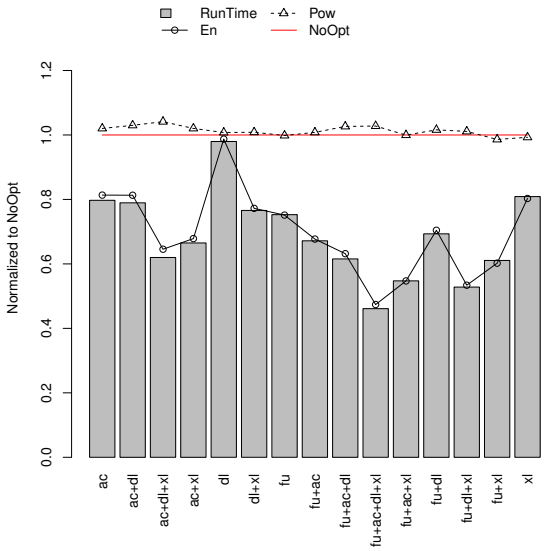

Figure 2. Execution time, power, and energy for each optimization.

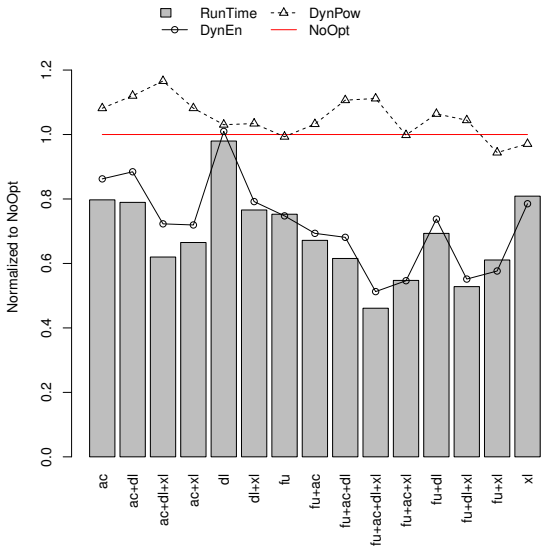

Figure 3. Execution time, dynamic energy, and dynamic power for each optimization.

a tradeoff is clear: improvements in execution time and memory power come at the expense of core power.

Figure 5 shows that all optimized versions except one result in less core and memory dynamic energy usage. In most cases memory energy is reduced more than core energy. In particular, memory energy is reduced the most in the code versions that have the shortest runtime and lowest overall energy usage.

Summarizing, understanding the effects of optimizations on power, energy, and execution time requires a detailed

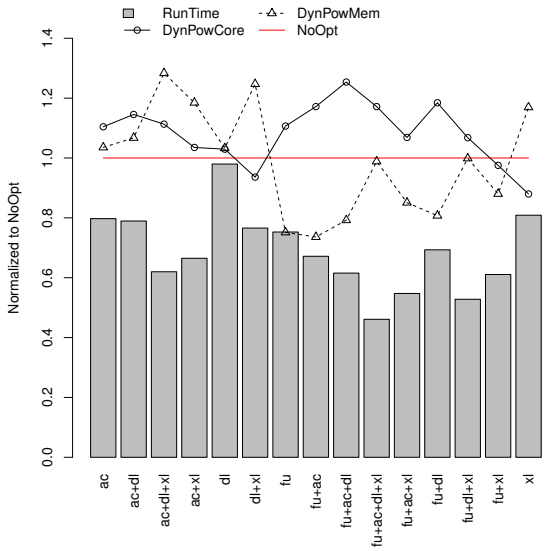

Figure 4. Execution time and dynamic power broken down by component for each optimization.

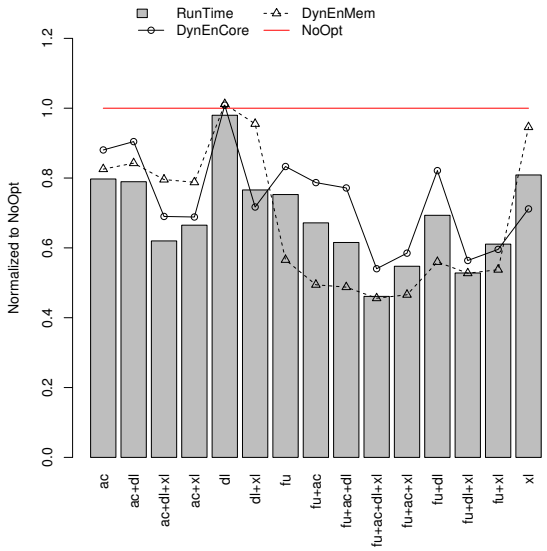

Figure 5. Execution time and dynamic energy broken down by component for each optimization.

analysis of various system levels including static and dynamic aspects as well as subsystem components such as the processor and memory.

V. Which Optimizations Impact Dynamic Power and Energy Usage and Why

In this section we investigate why code transformations impact dynamic power and energy usage by characterizing their impact at a subsystem level (i.e., processor and memory). We also correlate power and energy changes to performance events.

A. Power

Figure 6 compares dynamic memory power with memory bandwidth. We calculated memory bandwidth as the number of bytes moved to and from main memory per unit of time. This information is based on hardware events that count the number of cache line reads and writes to main memory. Figure 6 shows that dynamic memory power and bandwidth utilization have a nearly one to one relationship to each other. Therefore, reducing the rate at which bytes are moved directly decreases dynamic memory power consumption. For core power there is no single factor that correlates as well as for memory. Figure 7, however, shows that IPC and core power are partially correlated. Since the core domain contains the functional units, processor caches, and prefetch caches, it is not surprising that no single counter accounts for the entire variation.

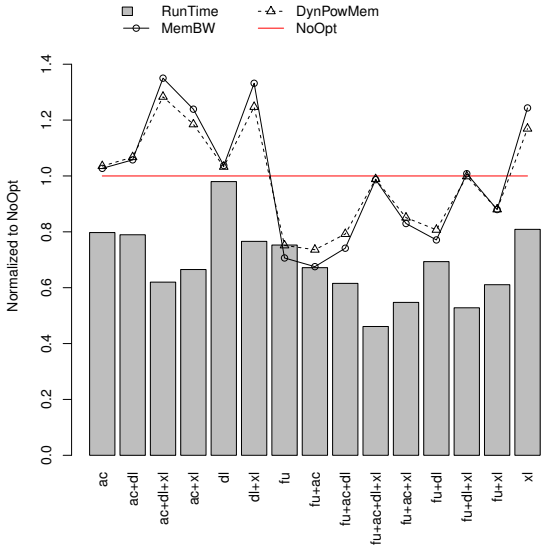

Figure 6. Memory bandwidth and dynamic memory power correlation.

Loop Fusion: Figures 6 and 7 show that all the codes with dynamic memory power usage less than or equal to the unoptimized code have the loop fusion optimization applied. However, fusion increases the power used by the core domain as shown by all pairs of optimized versions of LULESH that are identical other than whether loop fusion is applied, for example, xl and xl+fu. In all cases the version with loop fusion uses more core power. Loop fusion combines multiple accesses to the same data structure into a single loop reducing the number of times data is brought in from main memory. The result is fewer memory-bound loops and less data traffic per unit time, which results in lower dynamic memory power usage as seen in Figure 6. However,

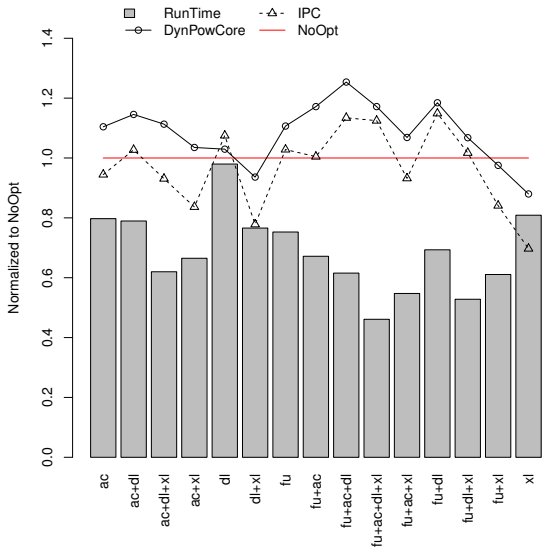

Figure 7. Instructions-per-cycle (IPC) and dynamic core power correlation.

the code becomes more compute intense, which results in a higher IPC and core cache usage shown in Figure 7.

XL Compiler: When comparing two codes that have identical optimizations except one was compiled with the XL compiler and one with GNU, for example, dl and dl+xl, we see in Figures 6 and 7 that dynamic memory power increases while dynamic core power and runtime decreases. The changes are related to the quality of code produced as opposed to changes in data motion because XL produces fewer integer instructions resulting in a lower IPC, core power, and runtime. The resulting code however, moves the same amount of data in less time resulting in a higher memory bandwidth utilization.

Data Layout and Allocation: In most cases the data layout and allocation optimizations increase memory power usage slightly, though in some combinations such as adding data layouts to allocation and the XL compiler (ac+dl+xl vs. ac+xl in Figure 4), the increase is more pronounced. Also, adding allocation to fusion (ac+fu vs. fu) reduces memory power slightly, which might imply that the arrays that were removed when fused are the ones that increase power usage when allocation is applied. Data layout and allocation also increase core power though the impact is never more than 10%, for example, ac vs. NoOpt and fu+ac vs. fu+ac+dl in Figure 4.

Overall, loop fusion is the most important optimization to reduce dynamic memory power consumption, while only the XL compiler reduces dynamic core power usage.

B. Energy

Figure 8 shows that dynamic memory energy and total data motion to and from memory are strongly correlated, with differences never exceeding 5%. Therefore, as with dynamic power the main way to reduce dynamic energy usage is to reduce data movement between memory and the processor. Figure 9 shows that dynamic core energy and the number of instructions executed are correlated. The gap is never larger than 20% and from the data we collected, instructions or subsets of instructions (e.g., floating-point) are the only clear counter that is directly proportional to dynamic core energy. Other factors, such as data motion from caches to the processor have no clear corrolations, but probably account for some or all of the difference.

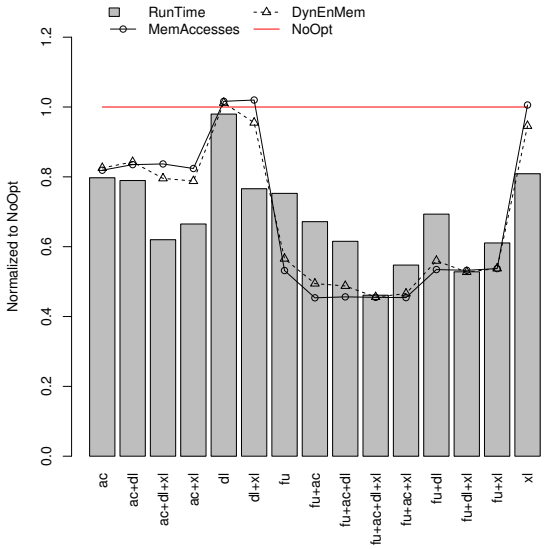

Figure 8. Memory accesses and dynamic memory energy correlation.

Loop Fusion: There is a group of eight codes in Figure 5 with the lowest memory energy use (60% or less than the baseline) that all employ the loop fusion (fu) optimization. These codes also move the fewest bytes from memory as shown in Figure 8. Loop fusion also significantly decreases dynamic core energy by up to 20% (see fu in Figure 9) because of fewer instructions executed, in particular loads and stores.

Allocation: Figure 8 shows that the set of eight codes implementing loop fusion consume the least amount of memory energy. Within these codes, the four with allocation applied (fu+ac, fu+ac+{dl,dl+xl,xl}), use the least amount of energy. Also, the codes with the least memory energy consumption that do not use loop fusion (ac, ac+{dl,dl+xl,xl} at about 80% the memory energy of the unoptimized code) all implement the allocation optimization. Thus, allocation is the second best optimization (after loop fusion) to reduce dynamic memory energy in LULESH. Note, however, that when both fusion and allocation are combined the effects are not additive and the combined code does not quite achieve the sum of the individual gains. Allocation also reduces core

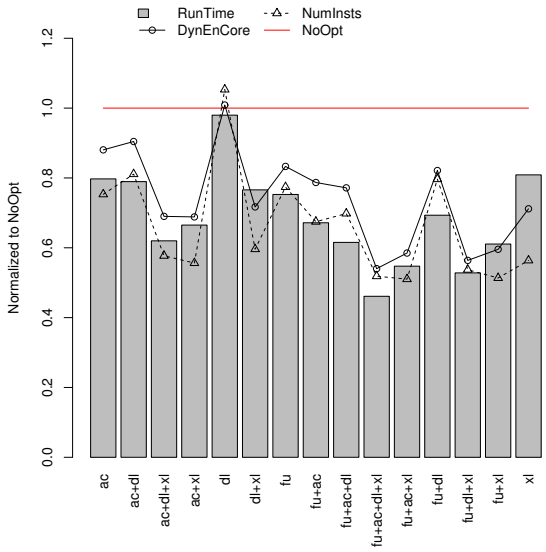

Figure 9. Number of instructions and dynamic core energy correlation.

dynamic energy usage by about 10% (e.g., ac in Figure 9). As with memory energy usage, the impact is smaller when combined with loop fusion since both remove the allocation of some arrays.

XL and Data Layouts: The XL compiler slightly reduces memory energy consumption (e.g., ac+dl vs. ac+dl+xl in Figure 8) while significantly decreasing core energy (e.g., xl in Figure 9). In fact the eight codes using the XL compiler are also the codes that use the least amount of dynamic core energy. Core energy is decreased due to code with significantly fewer instructions. Data motion from main memory (though loads from cache are reduced), however, is not changed. Data layout optimizations by themselves (xl vs. dl+xl in Figure 5) increase memory energy slightly though when combined with loop fusion (fu+xl vs. fu+xl+dl) decrease dynamic memory energy slightly. Data layout transformations also have a negligible impact on dynamic core energy use because they change how data is accessed more than the number of instructions used to access that data.

Overall, loop fusion and allocation are the most important optimizations to reduce memory energy usage. For core energy, using the XL compiler is most important, while loop fusion also has a significant impact on energy consumption.

C. Power, Energy, and Runtime Tradeoffs

Figure 3 shows that loop fusion (fu), using the XL compiler (xl), and the combination of both (fu+xl) reduce both dynamic power and energy usage, while also reducing runtime. Adding data allocation (ac) to those optimizations results in no significant changes in power usage, while reducing runtime and energy usage. However, all other optimization combinations, other than just applying data layouts, result in higher power and lower energy usage.

When tuning for runtime and (dynamic or total) energy usage, then applying all optimizations is best, at about a 10% power cost. Not performing allocation (fu+dl+xl) results in the second best runtime and energy usage (third best dynamic) when compared to the unoptimized code at a 5% increase in power. However, if there is a power cap equal to the baseline then not applying the data layout optimizations (fu+xl+ac) results in the third best runtime and ties for the second best dynamic energy (third best static) usage. Finally, when tuning for power then applying loop fusion and using the XL compiler results in the fourth best version in terms of energy and execution time.

There are tensions in trading off performance, power and energy on BG/Q. However, for LULESH we should always fuse loops and use the XL complier. After those two optimizations, the best choices depend on whether there is a power cap. With a power cap the best code will depend on how strict the power limit is and whether it uniformly affects the whole program.

VI. Discussion

In this paper we showed that tuning LULESH for energy on a BG/Q system is nearly identical to tuning for runtime. We also discovered that while tuning changes total power consumption, the impact is small. Therefore, we studied changes in dynamic power and energy consumption and observed a significant impact as a result of tuning. Furthermore, measuring the core and memory contributions separately provided key insights as to what optimizations are effective and why according to the significantly different amount of power consumed by each component. In addition, we correlated the changes in power and energy consumption to hardware counters and presented various runtime, energy, and power optimization tradeoffs.

There are a few caveats of analyzing *only dynamic* energy and power. If the power balance between components (e.g., memory and processor) stays the same in future systems and static power and energy are not reduced relative to dynamic power and energy, then tuning code for power will have marginal effects. And, tuning for energy will be similar to minimizing runtime. Because of the expected increase in leakage as voltages are reduced [7], system architectural changes may be necessary to allow effective tuning via program transformations. Some of these technologies, such as dynamic voltage and frequency scaling, are already features of current chips, while others, such as clock gating, are expected to appear in the near future.

Even if none of these technologies prove fruitful, we expect that the memory subsystem will consume a greater fraction of a machine's total power resulting in a greater amount of dynamic power. If systems were to evolve in this manner then there will be more oppertunity to tune for power and energy on them based on our expirimental data because

memory energy and power had a higher dynamic component. However, with main memory bandwidth increasing slower than floating-point capabilities, future systems will be more likely to run the memory system at its highest power state more frequently. For memory-bound HPC codes, this means that turning off cores when an application is in a memory-bound phase and reducing the memory bus speed when it is in a compute-bound phase may result in significant power and energy savings.

VII. Related work

We classify recent related work in the following areas: modeling power and energy in HPC; and analyzing performance, power, and energy tradeoffs. First, modeling is a useful vehicle to approximate power, energy, and temperature on systems that lack direct measurement capabilities. These models employ performance information to estimate the desired parameters [5], [16]. In addition, modeling is essential to predict power and energy on future systems. A few examples include modeling to estimate leakage energy on a cache hierarchy [7], modeling power and energy at finer granularities in terms of space and time [3], and modeling individual system components [17], [18]. Our work leverages existing power monitoring capabilities on BG/Q [2] and, similar to previous work, identifies the performance events that capture the driving force behind power and energy consumption on LULESH.

Second, a body of work described tradeoffs between power, energy, and execution time especially for compiler transformations. Wang et al. [6] used a polyhedral optimizer to generate multiple program variants with different optimizations including loop fusion, unrolling, tiling, and vectorization. Their results show a strong correlation between execution time and energy consumption. Balaprakash et al. [4] presents a multi-objective optimization framework to understand the trade-offs of performance, power, and energy. This formulation of objective functions can be used in auto-tuning environments with competing objectives. The authors demonstrated empirically that these tradeoffs exist. Tiwari et al. [5] uses compiler optimizations (loop tiling and unrolling) to validate their power and energy models of processor and memory components. Deshpande et al. [7] showed that certain compiler optimizations have a small effect on cache leakage.

This related work serves as a basis for *analyzing* why certain program transformations tradeoff power and energy, which is the focus of our work. Our contributions include a detailed analysis of power and energy consumption of several optimizations that have been successfully applied to explicit hydrodynamic codes in terms of performance. Unlike previous work, we provide insights that explain why a particular optimization tradeoffs power and energy the way it does. Our answers are based on a distinct analysis that considers static and dynamic elements separately at a subsystem-level granularity.

VIII. Conclusions and Future Work

In this paper we show that when tuning for power and energy the impact of program transformations is limited to the dynamic components of power and energy. We also show that the changes in dynamic power and energy usage for various system components is correlated with data motion and instructions executed. Of the optimizations analyzed, loop fusion and using the XL compiler improved all three objectives (power, energy, and execution time). However, data layout transformations and global allocations present tradeoffs in optimizing for the same three objectives.

Many HPC applications have multiple phases with different characteristics. In future work, we plan on characterizing program transformation optimizations enabled by changes in power states and by gating off parts of a chip without significantly decreasing performance. This work include analyzing compute and memory bound sections of LULESH and other representative HPC kernels and determining how fast power gating of memory and/or processor is needed between code regions to allow power savings at a marginal performance cost. We expect that this analysis of program transformations at a code-region granularity will allow more effective tuning by leveraging low power states and gating off technologies.

Acknowledgments

We would like to thank Paul Coteus, Hans Jacobson, Pradip Bose, Yutaka Sugawara, Ramon Bertran (IBM), Bronis de Supinski, Kim Cupps, Scott Futral, Dave Fox, and Teresa Kamakea (LLNL) for their help and support with IBM's high-resolution power monitoring infrastructure. We would also like to thank Bert Still and Martin Schulz (LLNL) for their contributions to refining the focus of this work and the anonymous reviewers for their encouraging feedback. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-648664.

References

[1] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *International Symposium on Low Power Electronics and Design*, ser. ISLPED'10. Austin, TX: ACM, Aug. 2010.

[2] R. Bertran, Y. Sugawara, H. M. Jacobson, A. Buyuktosunoglu, and P. Bose, "Application-level power and performance characterization and optimization on IBM Blue Gene/Q systems," *IBM Journal of Research and Development*, vol. 57, no. 1/2, 2013.

[3] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Enabling accurate power profiling of HPC applications on exascale systems," in *International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, Jun. 2013.

[4] P. Balaprakash, A. Tiwari, and S. M. Wild, "Multi objective optimization of HPC kernels for performance, power, and energy," in *International Workshop on Energy Efficient Supercomputing*, ser. E2SC'13. Denver, CO: ACM, Nov. 2013.

[5] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely, "Modeling power and energy usage of HPC kernels," in *Workshop on High-Performance, Power-Aware Computing*, ser. HPPAC'12. Shanghai, China: IEEE, May 2012.

[6] W. Wang, J. Cavazos, and A. Porterfield, "Energy auto-tuning using the polyhedral approach," in *Workshop on Polyhedral Compilation Techniques*, ser. IMPACT'14, Jan. 2014.

[7] A. Deshpande and J. Draper, "Leakage energy estimates for HPC applications," in *International Workshop on Energy Efficient Supercomputing*, ser. E2SC'13. ACM, Nov. 2013.

[8] I. Karlin, J. McGraw, E. Gallardo, J. Keasler, E. A. Leon, and B. Still, "Memory and parallelism exploration using the lulesh proxy application," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 1427–1428.

[9] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *IEEE International Parallel & Distributed Processing Symposium*, Boston, USA, May 2013.

[10] G. Gao, R. Olson, V. Sarkar, and R. Thekkath, "Collective loop fusion for array contraction." in *Workshop on Languages and Compilers for Parallel Computing*, Aug. 2004.

[11] I. Karlin, E. Jessup, and E. Silkensen, "Modeling the memory and performance impacts of loop fusion," *Journal of Computational Science*, 2011.

[12] K. Sharma, I. Karlin, J. Keasler, J. R. McGraw, and V. Sarkar, "User-Specified and Automatic Data Layout Selection for Portable Performance," Rice University, Houston, Texas, USA, Tech. Rep. TR13-03, April 2013.

[13] M. Gilge, "IBM system Blue Gene solution: Blue Gene/Q application development," *IBM Redbooks*, Jun. 2013.

[14] K. Yoshii, K. Iskra, R. Gupta, P. Beckman, V. Vishwanath, C. Yu, and S. Coghlan, "Evaluating power-monitoring capabilities on IBM Blue Gene/P and Blue Gene/Q," in *International Conference on Cluster Computing*. IEEE, Sep. 2012.

[15] S. Wallace, V. Vishwanath, S. Coghlan, Z. Lan, and M. E. Papka, "Measuring power consumption on IBM Blue Gene/Q," in *Workshop on High-Performance, Power-Aware Computing*, ser. HPPAC'13. Boston, MA: IEEE, May 2013.

[16] S. L. Song, K. Barker, and D. Kerbyson, "Unified performance and power modeling of scientific workloads," in *International Workshop on Energy Efficient Supercomputing*. ACM, 2013.

[17] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *International Symposium on Microarchitecture*, ser. MICRO-36. San Diego, CA: IEEE, Dec. 2003.

[18] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade, "Decomposable and responsive power models for multicore processors using performance counters," in *International Conference on Supercomputing*. ACM, 2010.